

# PatchTST for Time Series Forecasting: Original Results and My Single-Channel Experiments



Lalf · [Follow](#)

8 min read · May 17, 2023



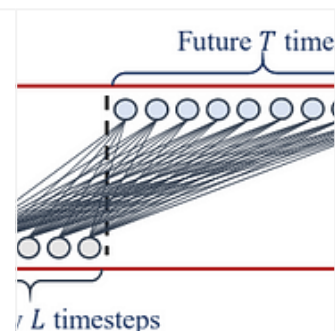
## Motivation

Transformers have become a popular choice for time series forecasting due to their ability to capture complex temporal relationships. However, recent studies have shown that simple deep neural network (DNN) models outperform Transformer models like FEDFormer and Informer in certain time series forecasting tasks[1]. I also wrote an article explaining the results of the experiment.

### Introduction of the paper: Are Transformers Effective for Time Series Forecasting?

Motivation

medium.com



To address this issue, a new Transformer architecture called PatchTST was introduced in a recent paper, “A TIME SERIES IS WORTH 64 WORDS: LONG-TERM FORECASTING WITH TRANSFORMERS”[2]. The PatchTST model incorporates two key concepts for time series forecasting: channel independence and patching.

Channel independence involves decomposing multichannel sequences into single channels before input to the model, allowing for greater flexibility and scalability in handling different types of data. Patching divides the input sequence into smaller parts, or patches, allowing the model to focus on local patterns and correlations.

In the original PatchTST paper, the authors evaluated the effectiveness of the model on multichannel time series forecasting tasks where both the input and output data are multichannel. To demonstrate the superiority of PatchTST over other models, the authors compared its performance with that of a simple DNN model, DLinear that outperformed Transformer models in a previous study.

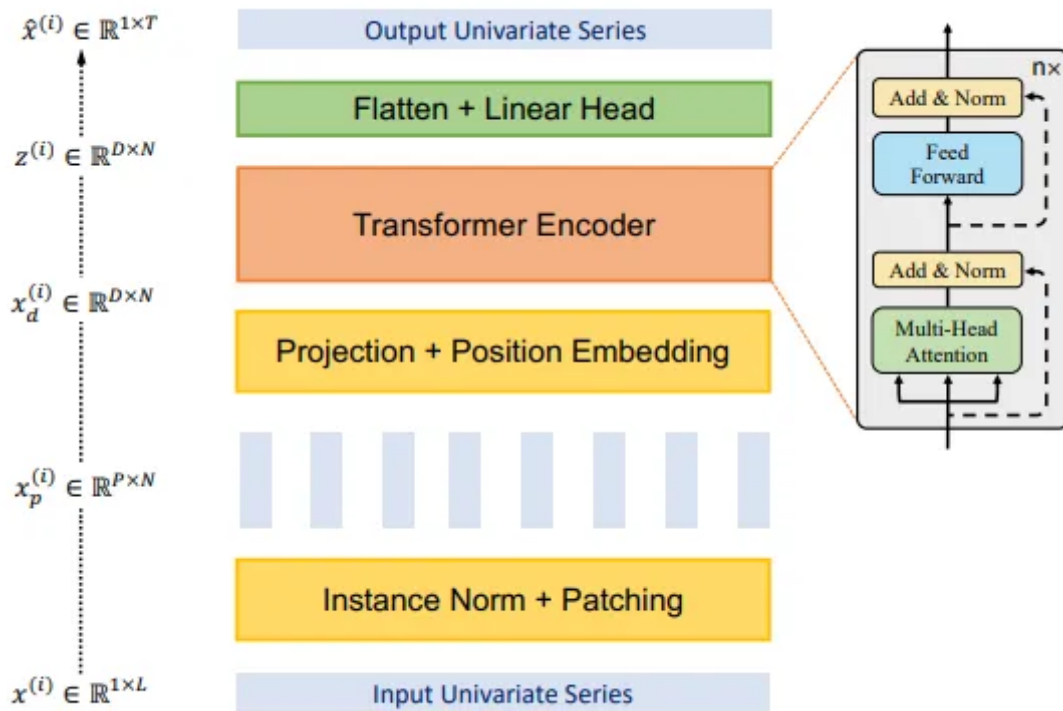
Inspired by this, I extended the application of PatchTST to a single-channel time series forecasting task with multichannel input data and single-channel output data. In this article, I will explore the PatchTST architecture and implementation of channel independence and patching. I will also discuss the results of my experiments and compare the performance of PatchTST with simple DNN models. My results demonstrate that PatchTST outperforms these models, further highlighting its potential for a wide range of time series forecasting applications.

## **Key concepts of PatchTST**

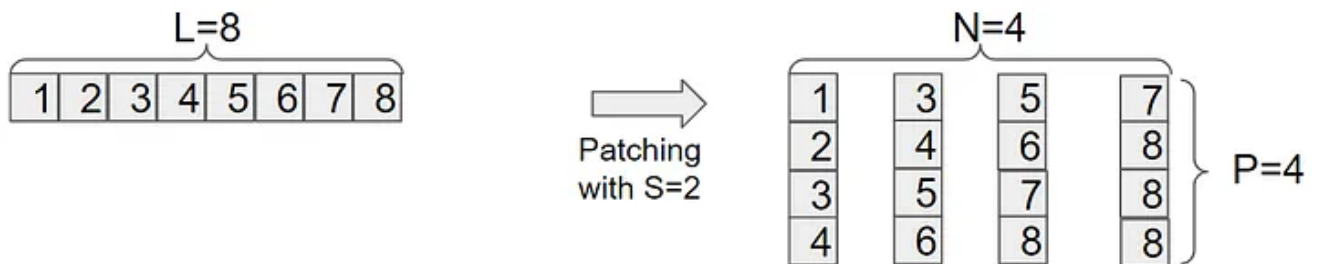
Mainly, there are two important concepts in the PatchTST model, “patching” and “channel independence”.

### **Patching**

Patching is a technique to alleviate the computational burden of self-attention. Rather than attending to every position in the sequence, the input sequence is partitioned into smaller sub-sequences known as patches. Self-attention is then computed between the patches. This approach enables the model to handle longer sequences while avoiding memory constraints and facilitating quicker inference. Additionally, patching enables the capture of localized semantic information that can not be available when using individual point-wise input tokens.



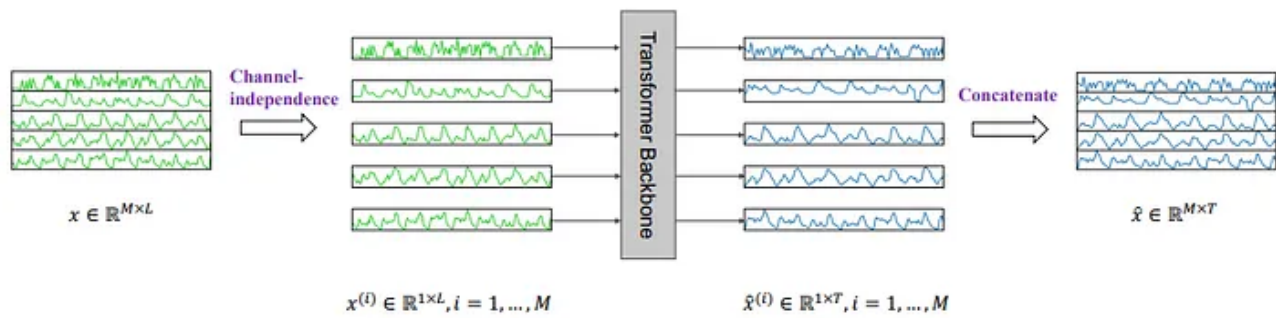
Conceptual diagram of the entire PatchTST and patching



By setting patch length(P) = 4 and stride(S) = 2 with sequence length(L) = 8, four patches are generated as shown in this figure.

### Channel independence

Channel independence, on the other hand, is a property of the PatchTST model that allows different channels of the input to be processed independently. In traditional transformers, the same set of attention weights is used for all channels, which limits the model's ability to capture fine-grained information in each channel. In contrast, the PatchTST model applies attention weights separately to each channel, allowing it to better capture the unique features and patterns in each channel.



(a) PatchTST Model Overview

Together, patching and channel independence make the PatchTST model a powerful tool for processing long sequences with multiple channels, such as long time series forecasting tasks. By dividing the input into patches and processing each channel independently, the model can efficiently capture complex patterns and relationships across the entire sequence.

## Results in the original paper

In the original paper, the authors conducted an extensive experiment to evaluate the performance of the PatchTST model on several datasets. They experimented with four patterns of output sequence lengths and compared the results with DLinear.



Models		PatchTST/64		PatchTST/42		DLinear		FEDformer		Autoformer		Informer		Pyraformer		LogTrans	
Metric		MSE	MAE	MSE	MAE	MSE	MAE	MSE	MAE	MSE	MAE	MSE	MAE	MSE	MAE	MSE	MAE
Weather	96	<b>0.149</b>	<b>0.198</b>	<u>0.152</u>	<u>0.199</u>	0.176	0.237	0.238	0.314	0.249	0.329	0.354	0.405	0.896	0.556	0.458	0.490
	192	<b>0.194</b>	<b>0.241</b>	<u>0.197</u>	<u>0.243</u>	0.220	0.282	0.275	0.329	0.325	0.370	0.419	0.434	0.622	0.624	0.658	0.589
	336	<b>0.245</b>	<b>0.282</b>	<u>0.249</u>	<u>0.283</u>	0.265	0.319	0.339	0.377	0.351	0.391	0.583	0.543	0.739	0.753	0.797	0.652
	720	<b>0.314</b>	<b>0.334</b>	<u>0.320</u>	<u>0.335</u>	0.323	0.362	0.389	0.409	0.415	0.426	0.916	0.705	1.004	0.934	0.869	0.675
Traffic	96	<b>0.360</b>	<b>0.249</b>	<u>0.367</u>	<u>0.251</u>	0.410	0.282	0.576	0.359	0.597	0.371	0.733	0.410	2.085	0.468	0.684	0.384
	192	<b>0.379</b>	<b>0.256</b>	<u>0.385</u>	<u>0.259</u>	0.423	0.287	0.610	0.380	0.607	0.382	0.777	0.435	0.867	0.467	0.685	0.390
	336	<b>0.392</b>	<b>0.264</b>	<u>0.398</u>	<u>0.265</u>	0.436	0.296	0.608	0.375	0.623	0.387	0.776	0.434	0.869	0.469	0.734	0.408
	720	<b>0.432</b>	<b>0.286</b>	<u>0.434</u>	<u>0.287</u>	0.466	0.315	0.621	0.375	0.639	0.395	0.827	0.466	0.881	0.473	0.717	0.396
Electricity	96	<b>0.129</b>	<b>0.222</b>	<u>0.130</u>	<u>0.222</u>	0.140	0.237	0.186	0.302	0.196	0.313	0.304	0.393	0.386	0.449	0.258	0.357
	192	<b>0.147</b>	<b>0.240</b>	<u>0.148</u>	<u>0.240</u>	0.153	0.249	0.197	0.311	0.211	0.324	0.327	0.417	0.386	0.443	0.266	0.368
	336	<b>0.163</b>	<b>0.259</b>	<u>0.167</u>	<u>0.261</u>	0.169	0.267	0.213	0.328	0.214	0.327	0.333	0.422	0.378	0.443	0.280	0.380
	720	<b>0.197</b>	<b>0.290</b>	<u>0.202</u>	<u>0.291</u>	0.203	0.301	0.233	0.344	0.236	0.342	0.351	0.427	0.376	0.445	0.283	0.376
ILI	24	<b>1.319</b>	<b>0.754</b>	<u>1.522</u>	<u>0.814</u>	2.215	1.081	2.624	1.095	2.906	1.182	4.657	1.449	1.420	2.012	4.480	1.444
	36	<u>1.579</u>	<u>0.870</u>	<b>1.430</b>	<b>0.834</b>	1.963	0.963	2.516	1.021	2.585	1.038	4.650	1.463	7.394	2.031	4.799	1.467
	48	<b>1.553</b>	<b>0.815</b>	<u>1.673</u>	<u>0.854</u>	2.130	1.024	2.505	1.041	3.024	1.145	5.004	1.542	7.551	2.057	4.800	1.468
	60	<b>1.470</b>	<b>0.788</b>	<u>1.529</u>	<u>0.862</u>	2.368	1.096	2.742	1.122	2.761	1.114	5.071	1.543	7.662	2.100	5.278	1.560
ETTh1	96	<b>0.370</b>	<u>0.400</u>	<u>0.375</u>	<b>0.399</b>	<u>0.375</u>	<b>0.399</b>	0.376	0.415	0.435	0.446	0.941	0.769	0.664	0.612	0.878	0.740
	192	<u>0.413</u>	0.429	0.414	<u>0.421</u>	<b>0.405</b>	<b>0.416</b>	0.423	0.446	0.456	0.457	1.007	0.786	0.790	0.681	1.037	0.824
	336	<b>0.422</b>	<u>0.440</u>	<u>0.431</u>	<b>0.436</b>	0.439	0.443	0.444	0.462	0.486	0.487	1.038	0.784	0.891	0.738	1.238	0.932
	720	<b>0.447</b>	0.468	<u>0.449</u>	<b>0.466</b>	0.472	0.490	0.469	0.492	0.515	0.517	1.144	0.857	0.963	0.782	1.135	0.852
ETTh2	96	<b>0.274</b>	<u>0.337</u>	<b>0.274</b>	<b>0.336</b>	0.289	0.353	0.332	0.374	0.332	0.368	1.549	0.952	0.645	0.597	2.116	1.197
	192	<u>0.341</u>	<u>0.382</u>	<b>0.339</b>	<b>0.379</b>	0.383	0.418	0.407	0.446	0.426	0.434	3.792	1.542	0.788	0.683	4.315	1.635
	336	<b>0.329</b>	<u>0.384</u>	<u>0.331</u>	<b>0.380</b>	0.448	0.465	0.400	0.447	0.477	0.479	4.215	1.642	0.907	0.747	1.124	1.604
	720	<b>0.379</b>	<b>0.422</b>	<b>0.379</b>	<b>0.422</b>	0.605	0.551	0.412	0.469	0.453	0.490	3.656	1.619	0.963	0.783	3.188	1.540
ETTm1	96	<u>0.293</u>	0.346	<b>0.290</b>	<b>0.342</b>	0.299	<u>0.343</u>	0.326	0.390	0.510	0.492	0.626	0.560	0.543	0.510	0.600	0.546
	192	<u>0.333</u>	0.370	<b>0.332</b>	<u>0.369</u>	0.335	<b>0.365</b>	0.365	0.415	0.514	0.495	0.725	0.619	0.557	0.537	0.837	0.700
	336	<u>0.369</u>	<u>0.392</u>	<b>0.366</b>	<u>0.392</u>	<u>0.369</u>	<b>0.386</b>	0.392	0.425	0.510	0.492	1.005	0.741	0.754	0.655	1.124	0.832
	720	<b>0.416</b>	<b>0.420</b>	<u>0.420</u>	0.424	0.425	<u>0.421</u>	0.446	0.458	0.527	0.493	1.133	0.845	0.908	0.724	1.153	0.820
ETTm2	96	<u>0.166</u>	<u>0.256</u>	<b>0.165</b>	<b>0.255</b>	0.167	0.260	0.180	0.271	0.205	0.293	0.355	0.462	0.435	0.507	0.768	0.642
	192	<u>0.223</u>	<u>0.296</u>	<b>0.220</b>	<b>0.292</b>	0.224	0.303	0.252	0.318	0.278	0.336	0.595	0.586	0.730	0.673	0.989	0.757
	336	<b>0.274</b>	<b>0.329</b>	<u>0.278</u>	<b>0.329</b>	0.281	0.342	0.324	0.364	0.343	0.379	1.270	0.871	1.201	0.845	1.334	0.872
	720	<b>0.362</b>	<b>0.385</b>	<u>0.367</u>	<b>0.385</b>	0.397	0.421	0.410	0.420	0.414	0.419	3.001	1.267	3.625	1.451	3.048	1.328

The best results are in bold and the second best are underlined.

The results showed that the PatchTST model outperformed the DLinear model significantly in terms of accuracy. Additionally, PatchTST was found to be superior to other Transformer models as well. The authors observed that the PatchTST model's ability to capture long-range dependencies and its channel independence property played a critical role in achieving better results. These findings suggest that PatchTST is a promising model for time series forecasting tasks, and its application can lead to significant improvements in prediction accuracy.

## My experiments with a single output channel

In this part, I will discuss my own experiments with PatchTST. For this, I chose the ETT dataset[3] and focused on a single output channel, specifically the Oil Temperature column. In contrast to the original paper where the output sequences were multichannel, I reconstructed PatchTST to output a single channel sequence. The reconstruction was a straightforward process and involved modifying the last fully connected layer.

# Implementation of the original final layer

```
class Flatten_Head(torch.nn.Module):
    def __init__(self, n_vars, nf, target_window, head_dropout=0):
        super().__init__()
        self.n_vars = n_vars
        self.flatten = torch.nn.Flatten(start_dim=-2)
        self.linear = torch.nn.Linear(nf, target_window)
        self.dropout = torch.nn.Dropout(head_dropout)

    def forward(self, x):
        x = self.flatten(x)
        x = self.linear(x)
        x = self.dropout(x)
        return x
```

# Implementation for single channel output

```
class Flatten_Head_For_Single_Output(torch.nn.Module):
    def __init__(self, n_vars, nf, target_window, head_dropout=0):
        super().__init__()
        self.n_vars = n_vars
        self.flatten = torch.nn.Flatten(start_dim=-3)
        self.linear = torch.nn.Linear(nf * n_vars, target_window)
        self.dropout = torch.nn.Dropout(head_dropout)

    def forward(self, x):
        x = self.flatten(x)
        x = self.linear(x)
        x = self.dropout(x)
        return x
```

#Implementation of PatchTST

```
class PatchTST(torch.nn.Module):

    def __init__(self, c_in, context_window, target_window, patch_len, stride,
                 n_layers=3, d_model=16, n_heads=4, d_k=None, d_v=None,
                 d_ff=128, attn_dropout=0.0, dropout=0.3, key_padding_mask=None,
                 padding_var=None, attn_mask=None, res_attention=True, pre_norm=False,
                 head_dropout = 0.0, padding_patch = "end",
                 revin = True, affine = False, subtract_last = False,
                 verbose=False, target_idx=-1, **kwargs):
        super().__init__()

        self.revin = revin
        if revin:
            self.revin_layer = RevIN(c_in, affine=affine, subtract_last=subtract_last)

        self.patch_len = patch_len
        self.stride = stride
        self.padding_patch = padding_patch
        patch_num = int((context_window - patch_len)/stride + 1)
```

```

        if padding_patch == "end":
            self.padding_patch_layer = torch.nn.ReplicationPad1d((0, stride))
            patch_num += 1

    self.backbone = TSTiEncoder(c_in, patch_num=patch_num, patch_len=patch_len,
                                n_layers=n_layers, d_model=d_model, n_heads=n_heads,
                                attn_dropout=attn_dropout, dropout=dropout, k=kernel_size,
                                attn_mask=attn_mask, res_attention=res_attention,
                                verbose=verbose, **kwargs)

    self.head_nf = d_model * patch_num
    self.n_vars = c_in

    self.head = Flatten_Head_For_Single_Output(self.n_vars, self.head_nf,

def forward(self, z):
    # instance norm
    if self.revin:
        z = self.revin_layer(z, "norm")
        z = z.permute(0,2,1)

    # patching
    if self.padding_patch == "end":
        z = self.padding_patch_layer(z)
    z = z.unfold(dimension=-1, size=self.patch_len, step=self.stride)
    z = z.permute(0,1,3,2)

    # model
    z = self.backbone(z)
    z = self.head(z)

    # denorm
    if self.revin:
        z = self.revin_layer(z, "denorm")
    return z

```

To evaluate the performance of the single channel PatchTST model, I compared it to both Linear and DLinear, which are implemented as follows.

```

#Linear model
class Linear(torch.nn.Module):
    def __init__(self, c_in, context_window, target_window):
        super().__init__()
        self.c_in = c_in
        self.context_window = context_window

```

```

self.target_window = target_window

self.flatten = torch.nn.Flatten(start_dim=-2)

self.linear = torch.nn.Linear(c_in * context_window, target_window)

def forward(self, x):
    # x: [bs x seq_len x nvars]
    x = self.flatten(x)          # x: [bs x seq_len * nvars]
    x = self.linear(x)           # x: [bs x target_window]
    return x

class moving_avg(torch.nn.Module):
    def __init__(self, kernel_size, stride):
        super().__init__()
        self.kernel_size = kernel_size
        self.avg = torch.nn.AvgPool1d(kernel_size=kernel_size, stride=stride,

    def forward(self, x):
        # padding on the both ends of time series
        front = x[:, 0:1, :].repeat(1, (self.kernel_size - 1) // 2, 1)
        end = x[:, -1:, :].repeat(1, (self.kernel_size - 1) // 2, 1)
        x = torch.cat([front, x, end], dim=1)
        x = self.avg(x.permute(0, 2, 1))
        x = x.permute(0, 2, 1)
        return x

class series_decomp(torch.nn.Module):
    def __init__(self, kernel_size):
        super().__init__()
        self.moving_avg = moving_avg(kernel_size, stride=1)

    def forward(self, x):
        moving_mean = self.moving_avg(x)
        res = x - moving_mean
        return res, moving_mean

#DLinear model
class DLinear(torch.nn.Module):
    def __init__(self, c_in, context_window, target_window):
        super().__init__()
        # Decomposition Kernel Size
        kernel_size = 25
        self.decomposition = series_decomp(kernel_size)
        self.flatten_Seasonal = torch.nn.Flatten(start_dim=-2)
        self.flatten_Trend = torch.nn.Flatten(start_dim=-2)

        self.Linear_Seasonal = torch.nn.Linear(c_in * context_window, target_
        self.Linear_Trend = torch.nn.Linear(c_in * context_window, target_wir

    def forward(self, x):

```



```

# x: [Batch, Input length, Channel]
seasonal_init, trend_init = self.decomposition(x)
seasonal_init = self.flatten_Seasonal(x)
trend_init = self.flatten_Trend(x)

seasonal_output = self.Linear_Seasonal(seasonal_init)
trend_output = self.Linear_Trend(trend_init)

x = seasonal_output + trend_output
return x

```

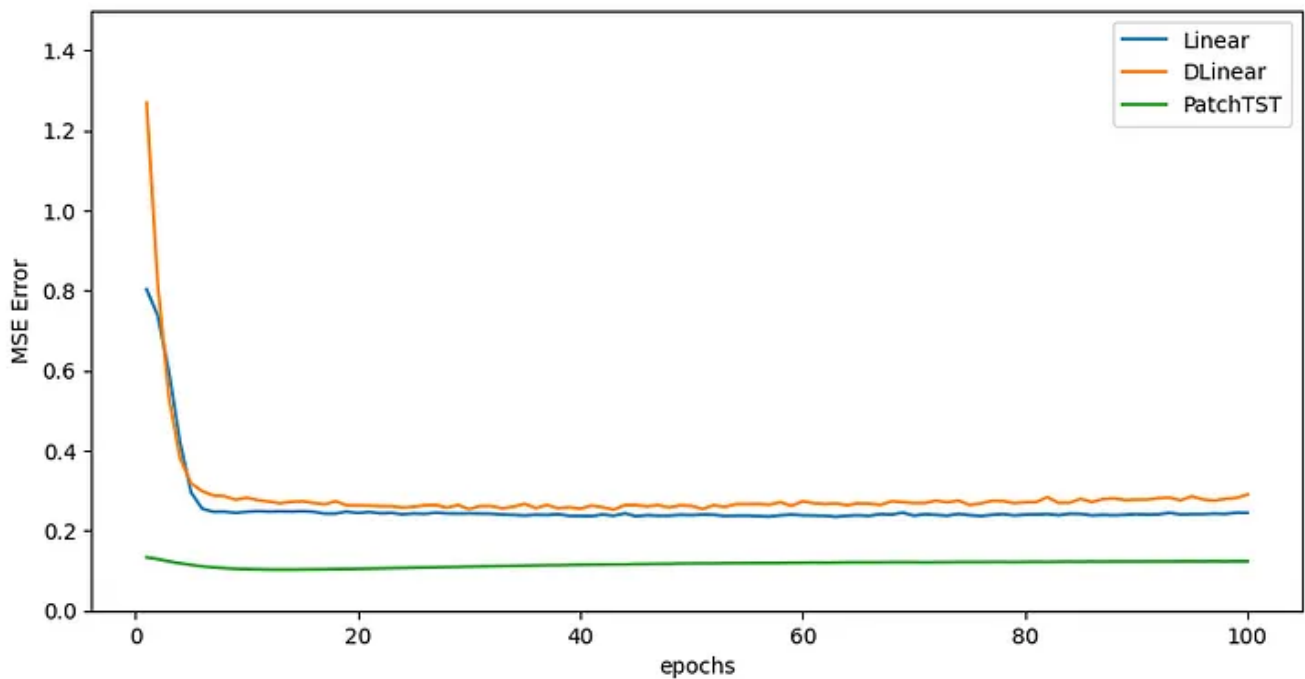
While I did not perform an extensive hyperparameter tuning, I did search for the optimal learning rate for each model.

The results of my experiments showed that even in the single output time series task, the single channel PatchTST model outperformed the Linear and DLinear models.

Model	PatchTST	Linear	DLinear
MSE	0.0621	0.0943	0.1087

MSE error for each model

In addition to comparing the performance of single channel PatchTST, Linear and DLinear on the ETT dataset, I also analyzed the learning process of each model. Using the validation dataset, I plotted the accuracy of each model over the course of the learning epochs.



The plot shows that PatchTST is not only more accurate but also more stable during the learning process. This indicates that PatchTST is able to learn from the data more effectively and consistently than Linear and DLinear. Overall, these findings suggest that PatchTST is a powerful and reliable tool for time series forecasting tasks, even when applied to single channel sequences.

## Conclusion

In conclusion, the PatchTST model offers a promising approach to time series forecasting tasks. By utilizing the concepts of patching and channel independence, it achieves better results than simple DNN models.

In this article, I have seen the key concepts of PatchTST and its superiority in multichannel time series forecasting tasks. Additionally, I presented the results of the original paper and my own experiments on the ETT dataset, demonstrating the effectiveness of PatchTST even in single output time series tasks.

Overall, PatchTST has the potential to revolutionize time series forecasting and should be considered as an alternative to traditional DNN models and other transformer-based models. Further research and experimentation can continue to explore its capabilities and applications in real-world scenarios.

The entire code used in these experiments is shown below, and I encourage everyone to try it.

patchtst\_single\_channel/PatchTST\_single\_channel\_exp.ipynb  
at main · ...

You can't perform that action at this time. You signed in with  
another tab or window. You signed out in another tab or...

github.com

n/  
t\_single\_channel

0 Issues 0 Stars 0 Forks

## References

[1] A. Zeng, M. Chen, L. Zhang, and Q. Xu, "Are Transformers Effective for Time Series Forecasting?," in Proceedings of the AAAI Conference on Artificial Intelligence, 2023.

[2] Y. Nie, N. H. Nguyen, P. Sinthong, and J. Kalagnanam, "A Time Series is Worth 64 Words: Long-term Forecasting with Transformers," in Proceedings of the International Conference on Learning Representations, 2023.

[3] H. Zhou, S. Zhang, J. Peng, S. Zhang, J. Li, H. Xiong, and W. Zhang, "Informer: Beyond Efficient Transformer for Long Sequence Time-Series Forecasting," in Proceedings of the 35th AAAI Conference on Artificial Intelligence, AAAI 2021, Virtual Conference, vol. 35, no. 12, pp. 11106–11115, 2021.

Deep Learning

Machine Learning

Data Science

Time Series Forecasting

Transformers

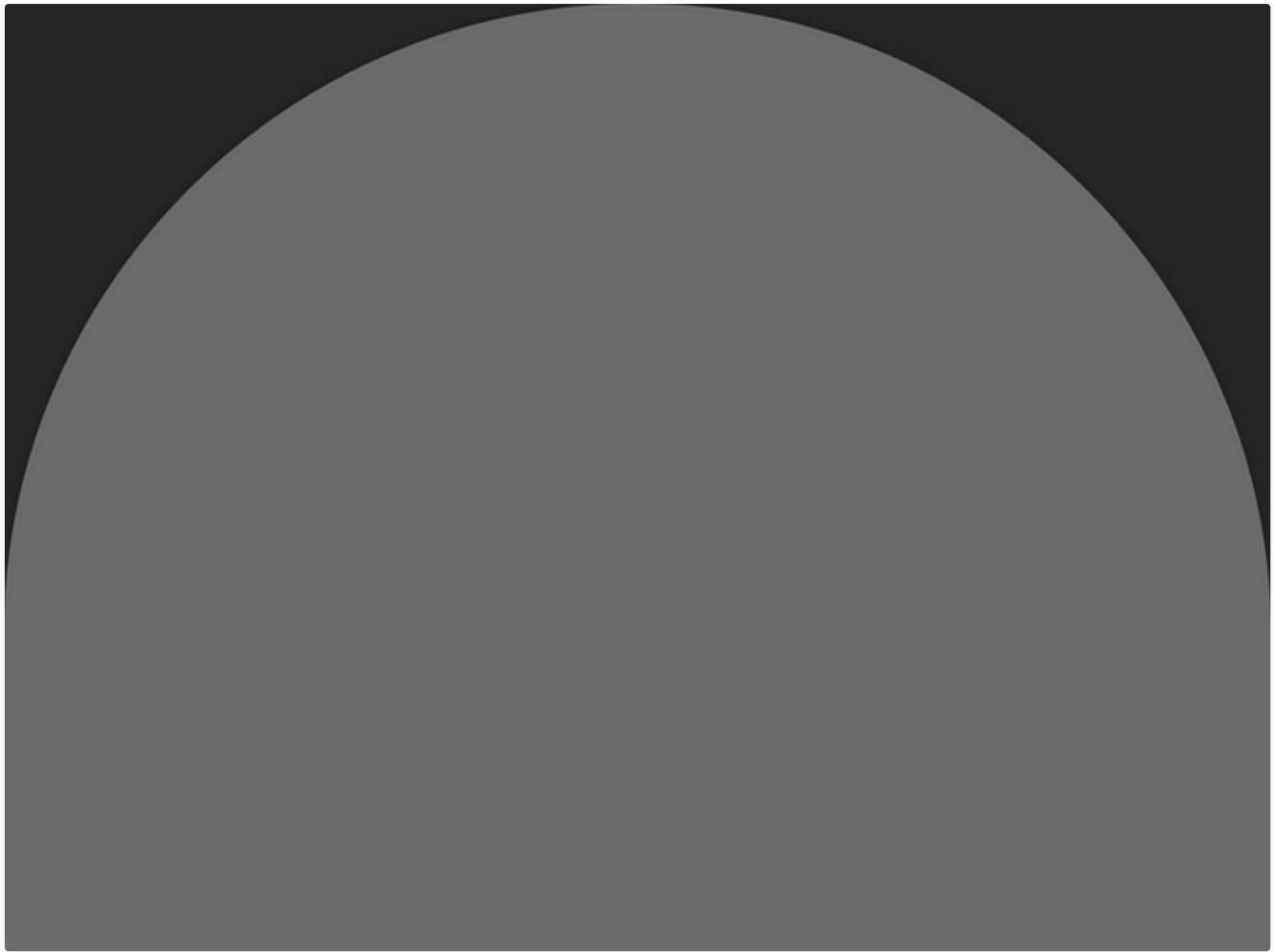


## Written by Lalf

38 Followers

Data Scientist for a global e-commerce company

## More from Lalf



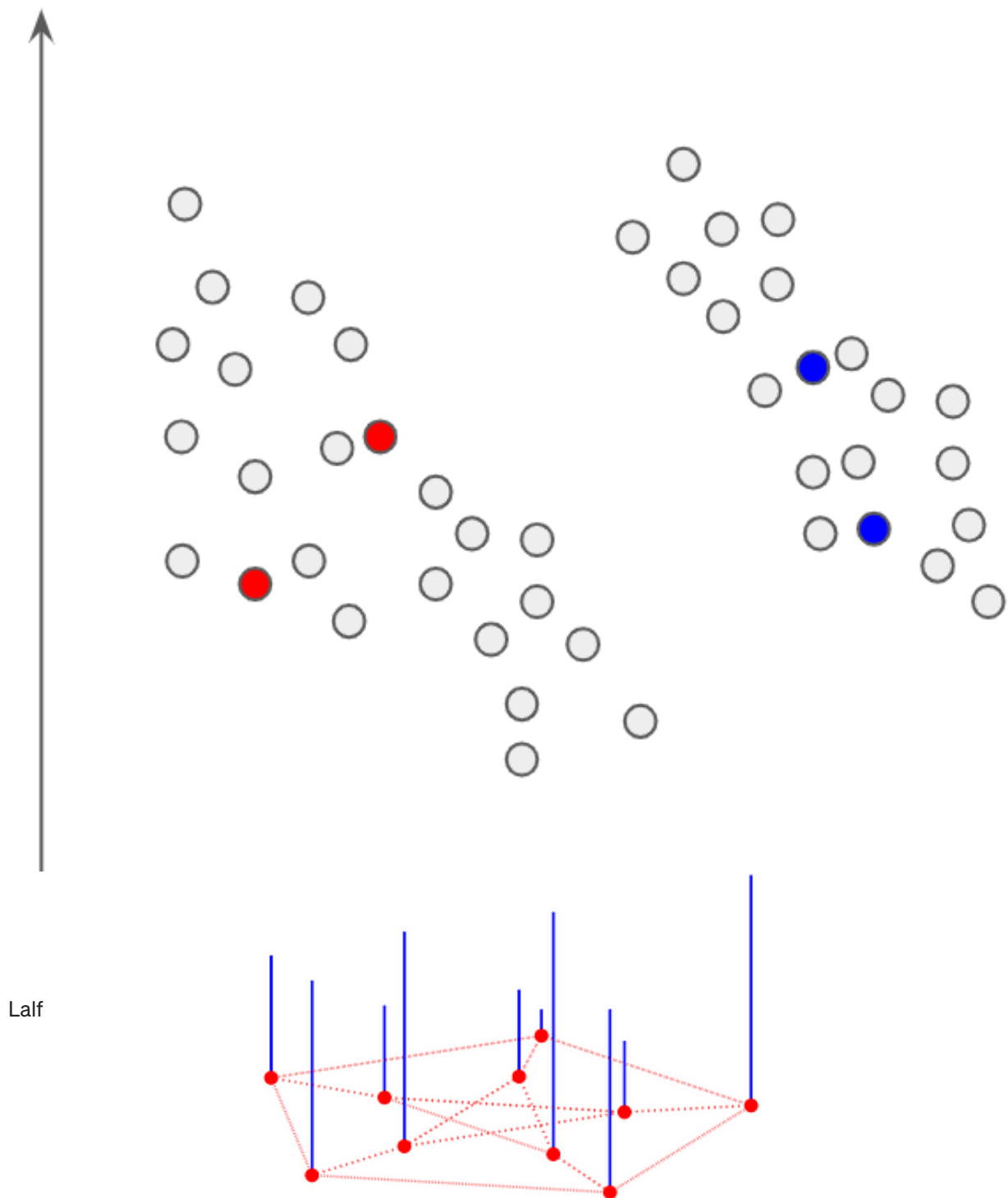
Lalf

## Random Sampling Tips for Google BigQuery

Why do you need random sampling?

2 min read · Dec 29, 2022





Lalf

Fig. 1. A random positive graph signal on the vertices of the Petersen graph. The height of each blue bar represents the signal value at the vertex where the bar originates.

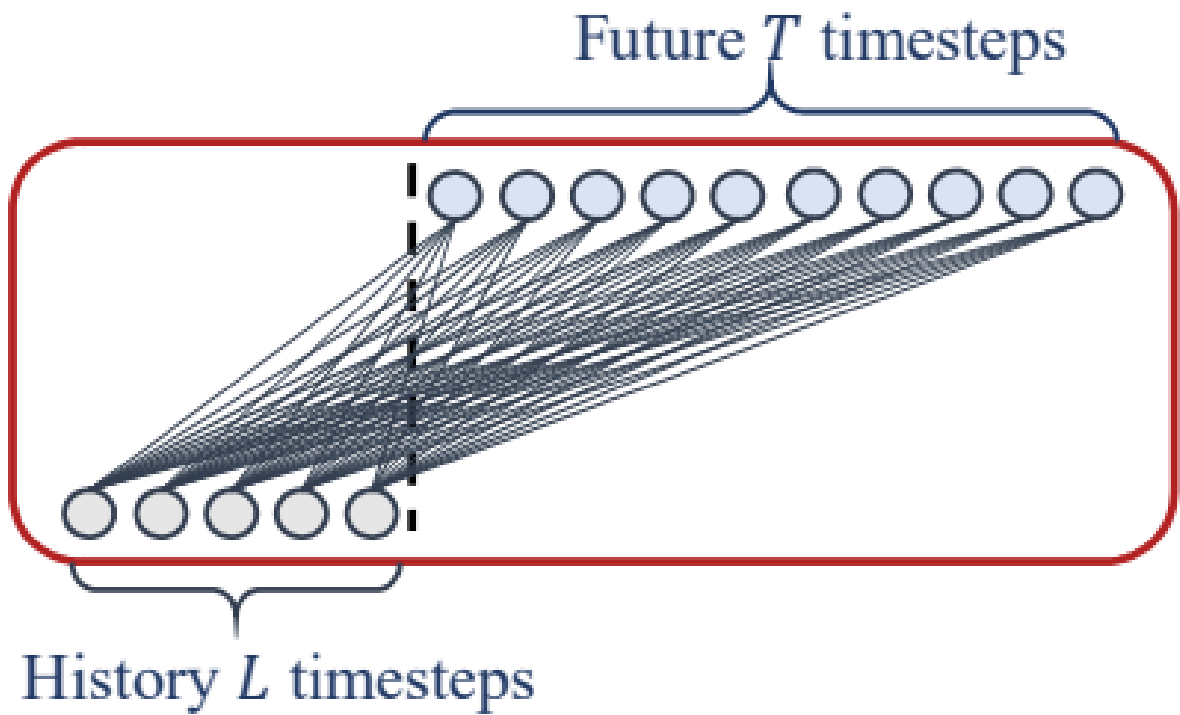


Lalf

## From Graph Fourier Transform to Graph Convolution

About Graph Neural Network

8 min read · Aug 16, 2023



Lalf

## Introduction of the paper: Are Transformers Effective for Time Series Forecasting?

Motivation

6 min read · May 8, 2023



See all from Lalf

Recommended from Medium





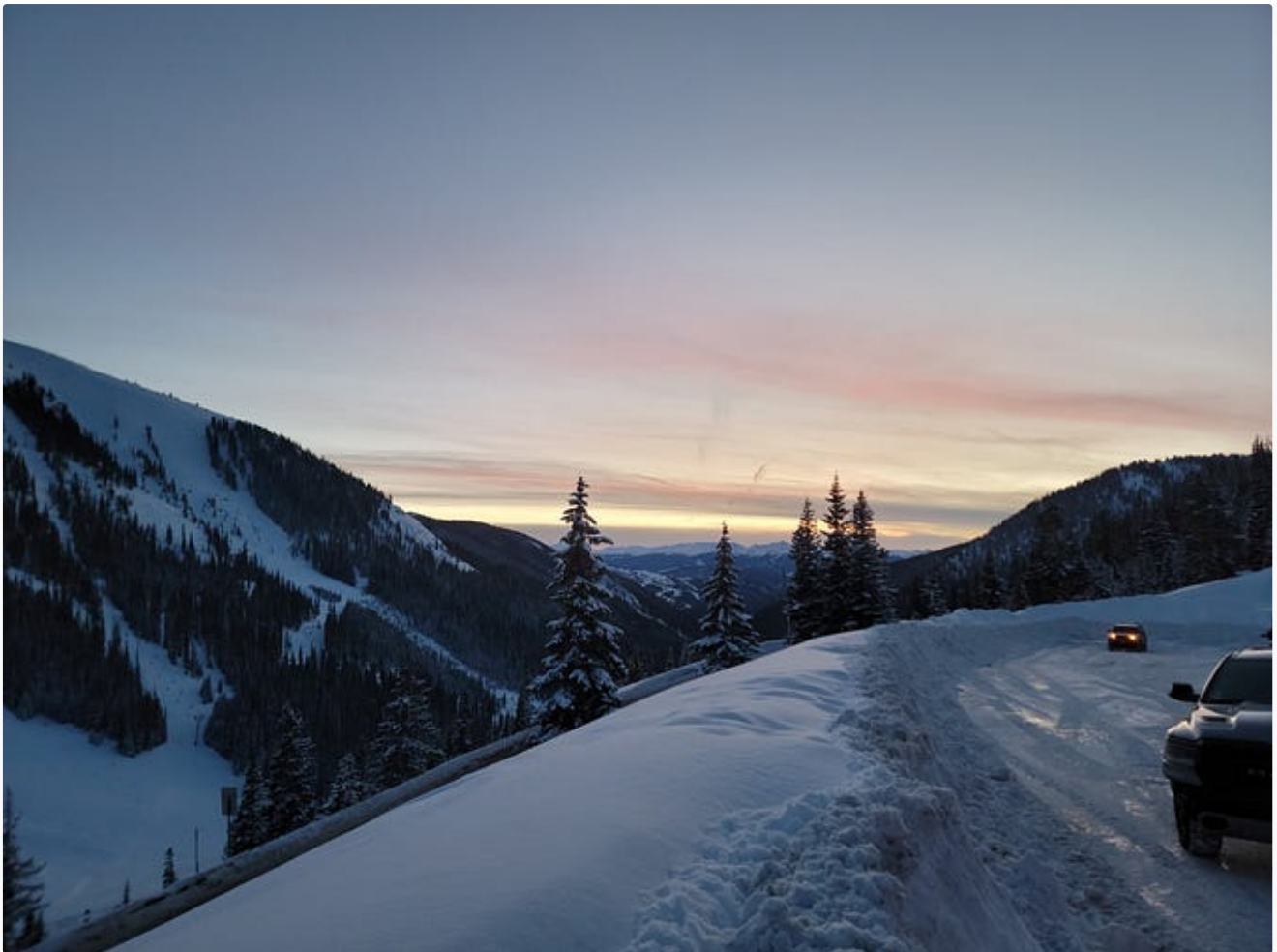
 Marco Peixeiro  in Towards Data Science

## PatchTST: A Breakthrough in Time Series Forecasting

From theory to practice, understand the PatchTST algorithm and apply it in Python alongside N-BEATS and N-HiTS

★ · 10 min read · Jun 20, 2023





## Lists



Isaac Godfried in Deep Data Science



### Predictive Modeling w/ Python in Deep Learning for Time Series Forecasting/Classification

20 stories · 1041 saves



### Practical Guides to Machine Learning

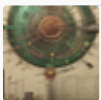
ies forecasting had its OOP moment! Have transformers overcome their  
in time series and how they out perform...

128 stories · 278 saves



### Natural Language Processing


1327 stories · 813 saves



### data science and AI

40 stories · 114 saves



 samuel chazy in Artificial Intelligence in Plain English

## Forecasting sales using Meta's Neural Prophet

A powerful tool for time series forecasting combining neural networks with the intuitive modeling of traditional forecasting methods.

10 min read · Oct 25, 2023











Jesus Rodriguez

## Google Just Built a Foundation Model for Zero-Shot Time Series Forecasting

A decoder-only transformer for predictions in time series data.

5 min read · Feb 5, 2024



---

See more recommendations